LAB ASSIGNMENT 11.3

Program : B. Tech (CSE)

Specialization : AIML

Course Title : AI Assisted coding

Semester : III

Academic Session : 2025-2026

Name of Student : K.Pardhasaradhi

Enrollment No : 2403A52001

Batch No : 01

Date : 08-10-2025

- **ACTIVITY**

Task Description #1 – Stack class implementation

Task: Ask AI to implement a stack class with push(), pop(), peek() and is_empty() methods.

Prompt: Create python class implementation for a stack data structure, including methods for adding elements (push), removing elements (pop), viewing the top element without removing it (peek), and checking if the stack is empty (is_empty)?"

Code :

```
]
1m
```

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
        print(f"Pushed: {item}")

    def pop(self):
        if not self.is_empty():
            item = self.items.pop()
            print(f"Popped: {item}")
            return item
        else:
            print("Stack is empty. Cannot pop.")
            return None

    def peek(self):
        if not self.is_empty():
            item = self.items[-1]
            print(f"Peeked: {item}")
            return item
        else:
            print("Stack is empty. No item to peek.")
```

```
        else:
            print("Stack is empty. No item to peek.")
            return None

    def is_empty(self):
        return len(self.items) == 0

# Example usage with user input
stack = Stack()

while True:
    action = input("Enter action (push, pop, peek, is_empty, quit): ").lower()

    if action == "quit":
        break
    elif action == "push":
        item = input("Enter item to push: ")
        stack.push(item)
    elif action == "pop":
        stack.pop()
    elif action == "peek":
        stack.peek()
    elif action == "is_empty":
        print(f"Is stack empty? {stack.is_empty()}")
    else:
```

```
    elif action == "push":
        item = input("Enter item to push: ")
        stack.push(item)
    elif action == "pop":
        stack.pop()
    elif action == "peek":
        stack.peek()
    elif action == "is_empty":
        print(f"Is stack empty? {stack.is_empty()}")
    else:
        print("Invalid action. Please try again.")
```

```
Enter action (push, pop, peek, is_empty, quit): push
Enter item to push: book
Pushed: book
Enter action (push, pop, peek, is_empty, quit): is_empty
Is stack empty? False
Enter action (push, pop, peek, is_empty, quit): quit
```

Observation :

- **Implementation:** The stack is implemented using a Python list (self.items). Lists are a suitable choice for this as they provide dynamic resizing and efficient append/pop operations from the end.
- **LIFO Principle:** The push and pop methods correctly follow the Last-In, First-Out (LIFO) principle that defines a stack. push adds to the end of the list, and pop removes from the end.
- **Error Handling:** The pop and peek methods include basic error handling to check if the stack is empty before attempting an operation that would fail. This prevents potential IndexError exceptions.
- **is_empty Method:** The is_empty method provides a clear and simple way to check the state of the stack by checking the length of the underlying list.
- **User Interaction:** The example usage includes a while loop and input() calls to allow the user to interact with the stack by choosing actions and providing input. This makes the code easy to test and understand.

- **Clear Output:** The methods include print statements to show the action being performed (e.g., "Pushed: item", "Popped: item"), which is helpful for demonstrating the stack's behavior.

Task Description #2 – Queue Implementation

Task: Use AI to generate a Queue class with enqueue(), dequeue(), and is_empty().

Prompt : create a Python code for a Queue data structure, including functions to add elements to the rear (enqueue), remove elements from the front (dequeue), and determine if the queue is empty (is_empty)?"

Code :

```
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)
        print(f"Enqueued: {item}")

    def dequeue(self):
        if not self.is_empty():
            item = self.items.pop(0)  # Remove from the front
            print(f"Dequeued: {item}")
            return item
        else:
            print("Queue is empty. Cannot dequeue.")
            return None

    def is_empty(self):
        return len(self.items) == 0

# Example usage with user input
queue = Queue()
```

```
while True:
    action = input("Enter action (enqueue, dequeue, is_empty, quit): ").lower()

    if action == "quit":
        break
    elif action == "enqueue":
        item = input("Enter item to enqueue: ")
        queue.enqueue(item)
    elif action == "dequeue":
        queue.dequeue()
    elif action == "is_empty":
        print(f"Is queue empty? {queue.is_empty()}")
    else:
        print("Invalid action. Please try again.")
```

Output :

```
Enter action (enqueue, dequeue, is_empty, quit): enqueue
Enter item to enqueue: pen
Enqueued: pen
Enter action (enqueue, dequeue, is_empty, quit): dequeue
Dequeued: pen
Enter action (enqueue, dequeue, is_empty, quit): quit
```

Observation :

- **Implementation**: Like the Stack, the Queue is implemented using a Python list (self.items).
- **FIFO Principle:** The enqueue and dequeue methods correctly follow the First-In, First-Out (FIFO) principle that defines a queue. enqueue adds to the end of the list, while dequeue removes from the beginning of the list using pop(0).
- **pop(0) Efficiency:** While using pop(0) on a Python list works for a queue, it's important to note that removing from the beginning of a list can be less efficient for very large lists compared to adding and removing from the end (as in the Stack). For performance-critical applications with large queues, using Python's collections.deque might be a better choice as it's optimized for adding and removing from both ends.
- **Error Handling:** Similar to the Stack, the dequeue method includes error handling to check if the queue is empty before attempting to remove an item.

- **is_empty Method:** The is_empty method is a simple and effective way to check if the queue contains any elements.
- **User Interaction:** The example usage provides a clear way for the user to interact with the queue and test its methods through input prompts.
- **Clear Output:** The print statements in the methods help visualize the process of enqueuing and dequeuing items.

Task Description #3 – Linked List Implementation

Task: Ask AI to create a singly linked list with insert_at_end(),

insert_at_beginning(), and display().

Prompt : create Python code for a singly linked list data structure that includes functions to add nodes at the end (insert_at_end), add nodes at the beginning (insert_at_beginning), and print the elements of the list (display)?".

Code :

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        print(f"Inserted {data} at the beginning.")

    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            print(f"Inserted {data} at the end.")
            return
```

```
        if self.head is None:
            self.head = new_node
            print(f"Inserted {data} at the end.")
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
        print(f"Inserted {data} at the end.")

    def display(self):
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        print("Linked List:", elements)


# Example usage with user input
linked_list = SinglyLinkedList()

while True:
    action = input("Enter action (insert_beginning, insert_end, display, quit): ").lower()

    if action == "quit":
        break
    elif action == "insert_beginning":
        data = input("Enter data to insert at beginning: ")
        linked_list.insert_at_beginning(data)
    elif action == "insert_end":
        data = input("Enter data to insert at end: ")
        linked_list.insert_at_end(data)
    elif action == "display":
        linked_list.display()
    else:
        print("Invalid action. Please try again.")
```

Output :

```
Enter action (insert_beginning, insert_end, display, quit): insert_beginning
Enter data to insert at beginning: Assisted coding
Inserted Assisted coding at the beginning.
Enter action (insert_beginning, insert_end, display, quit): display
Linked List: ['Assisted coding']
Enter action (insert_beginning, insert_end, display, quit): quit
```

Observation :

- **Node Class:** The code correctly defines a Node class as a building block for the linked list. Each node stores data and a reference to the next node in the sequence.
- **Singly Linked:** The implementation is for a singly linked list because each node only has a reference to the next node, not the previous one.
- **Head Pointer:** The SinglyLinkedList class maintains a head pointer, which references the first node in the list. This is the entry point for traversing or modifying the list.
- **insert_at_beginning:** This method is efficient. It creates a new node, sets its next pointer to the current head, and then updates the list's head to point to the new node. This takes constant time (O(1)).
- **insert_at_end:** This method requires traversing the list from the head to the last node to append the new node. In the worst case (a long list), this takes time proportional to the number of nodes (O(n)). An edge case for an empty list is handled correctly.
- **display:** This method also requires traversing the list from the head to collect and print the data of each node. This operation takes time proportional to the number of nodes (O(n)).
- **User Interaction**: The example usage with the while loop and input() allows for easy testing and demonstration of the linked list's functionality.
- **Clear Output:** The print statements in the methods provide feedback on the operations being performed.

Task Description #4 – Binary Search Tree (BST)
Task: Ask AI to generate a simple BST with insert() and inorder_traversal().

Prompt : create Python code for a basic Binary Search Tree (BST) data structure, including a method to insert new nodes (insert) and a method to perform an in-order traversal of the tree (inorder_traversal)?"

Code :

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
            print(f"Inserted {key} as the root.")
        else:
            self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = Node(key)
                print(f"Inserted {key} to the left of {node.key}.")
            else:
```

```
                node.left = Node(key)
                print(f"Inserted {key} to the left of {node.key}.")
            else:
                self._insert_recursive(node.left, key)
        elif key > node.key:
            if node.right is None:
                node.right = Node(key)
                print(f"Inserted {key} to the right of {node.key}.")
            else:
                self._insert_recursive(node.right, key)
        else:
            print(f"Key {key} already exists in the BST.")

    def inorder_traversal(self):
        elements = []
        self._inorder_recursive(self.root, elements)
        print("In-order Traversal:", elements)
        return elements

    def _inorder_recursive(self, node, elements):
        if node:
            self._inorder_recursive(node.left, elements)
            elements.append(node.key)
            self._inorder_recursive(node.right, elements)
```

```
# Example usage with user input
bst = BST()

while True:
    action = input("Enter action (insert, inorder_traversal, quit): ").lower()

    if action == "quit":
        break
    elif action == "insert":
        try:
            key = int(input("Enter key to insert: "))
            bst.insert(key)
        except ValueError:
            print("Invalid input. Please enter an integer key.")
    elif action == "inorder_traversal":
        bst.inorder_traversal()
    else:
        print("Invalid action. Please try again.")
```

Output :

```
Enter action (insert, inorder_traversal, quit): insert
Enter key to insert: 10
Inserted 10 as the root.
Enter action (insert, inorder_traversal, quit): inorder_traversal
In-order Traversal: [10]
Enter action (insert, inorder_traversal, quit): quit
```

Observation :

- **Node Class:** The code defines a Node class to represent the nodes in the tree. Each node stores a key (the value) and references to its left and right children.
- **Root Node:** The BST class maintains a root pointer, which is the starting point of the tree.
- **insert Method:** The insert method handles the logic for adding new nodes while maintaining the BST property (left child key < parent key < right child key). It uses a recursive helper function _insert_recursive to traverse the tree and find the correct position for the new key.
- **Recursive Insertion:** The _insert_recursive function demonstrates the recursive nature of inserting into a BST. It compares the new key with the current node's key to decide whether to go left or right until an empty spot is found.

- **Handling Duplicates:** The code includes a check to see if the key already exists in the tree and prints a message if it does, preventing duplicate keys.
- **inorder_traversal Method**: The inorder_traversal method performs an in-order traversal of the tree, which visits the left subtree, then the current node, and then the right subtree. This traversal method is commonly used for BSTs because it visits the nodes in ascending order of their keys
- **Recursive Traversal:** The _inorder_recursive function is a recursive helper that implements the in-order traversal logic.